

DIGITAL DESIGN AND COMPUTER ARCHITECTURE (252-0028-00L), SPRING 2020
OPTIONAL HW 3: ISA, MICROARCHITECTURE, AND PERFORMANCE EVALUATION

SOLUTIONS

Instructor: Prof. Onur Mutlu

TAs: Mohammed Alser, Rahul Bera, Can Firtina, Juan Gomez-Luna, Jawad Haj-Yahya, Hasan Hassan, Konstantinos Kanellopoulos, Lois Orosa, Jisung Park, Geraldo De Oliveira Junior, Minesh Patel, Giray Yaglikci

Released: Tuesday, March 31, 2020

1 Big versus Little Endian Addressing

Consider the 32-bit hexadecimal number 0xcafe2b3a.

1. What is the binary representation of this number in *little endian* format? Please clearly mark the bytes and number them from low (0) to high (3).

3a	2b	fe	ca
0	1	2	3

2. What is the binary representation of this number in *big endian* format? Please clearly mark the bytes and number them from low (0) to high (3).

ca	fe	2b	3a
0	1	2	3

2 The MIPS ISA

2.1 Warmup: Computing a Fibonacci Number

The Fibonacci number F_n is recursively defined as

$$F(n) = F(n-1) + F(n-2),$$

where $F(1) = 1$ and $F(2) = 1$. So, $F(3) = F(2) + F(1) = 1 + 1 = 2$, and so on. Write the MIPS assembly for the `fib(n)` function, which computes the Fibonacci number $F(n)$:

```
int fib(int n)
{
    int a = 0;
    int b = 1;
    int c = a + b;
    while (n > 1) {
        c = a + b;
        a = b;
        b = c;
        n--;
    }
    return c;
}
```

Remember to follow MIPS calling convention and its register usage (just for your reference, you may not need to use all of these registers):

- The argument `n` is passed in register `$4`.
- The result (i.e., `c`) should be returned in `$2`.
- `$8` to `$15` are caller-saved temporary registers.
- `$16` to `$23` are callee-saved temporary registers.
- `$29` is the stack pointer register.
- `$31` stores the return address.

Note: A summary of the MIPS ISA is provided at the end of this handout.

NOTE: More than one correct solution exists, this is just one potential solution.

```
fib:
addi $sp, $sp, -16 // allocate stack space
sw  $16, 0($sp)    // save r16
add  $16, $4, $0   // r16 for arg n
sw  $17, 4($sp)    // save r17
add  $17, $0, $0   // r17 for a, init to 0
sw  $18, 8($sp)    // save r18
addi $18, $0, 1    // r18 for b, init to 1
sw  $31, 12($sp)   // save return address
add  $2, $17, $18  // c = a + b

branch:
slti $3, $16, 2    // use r3 as temp
bne  $3, $0, done
add  $2, $17, $18  // c = a + b
add  $17, $18, $0  // a = b
add  $18, $2, $0   // b = c
addi $16, $16, -1  // n = n - 1
j    branch

done:
lw  $31, 12($sp)   // restore r31
lw  $18, 8($sp)    // restore r18
lw  $17, 4($sp)    // restore r17
lw  $16, 0($sp)    // restore r16
addi $sp, $sp, 16  // restore stack pointer
jr  $31            // return to caller
```

2.2 MIPS Assembly for REP MOVSB

MIPS is a simple ISA. Complex ISAs—such as Intel’s x86—often use one instruction to perform the function of many instructions in a simple ISA. Here you will implement the MIPS equivalent for a single Intel x86 instruction, REP MOVSB, which is specified as follows.

The REP MOVSB instruction uses three fixed x86 registers: ECX (count), ESI (source), and EDI (destination). The “repeat” (REP) prefix on the instruction indicates that it will repeat ECX times. Each iteration, it moves one byte from memory at address ESI to memory at address EDI, and then increments both pointers by one. Thus, the instruction copies ECX bytes from address ESI to address EDI.

- (a) Write the corresponding assembly code in MIPS ISA that accomplishes the same function as this instruction. You can use any general purpose register. Indicate which MIPS registers you have chosen to correspond to the x86 registers used by REP MOVSB. Try to minimize code size as much as possible.

Assume: \$1 = ECX, \$2 = ESI, \$3 = EDI

```
beq    $1, $0, AfterLoop    // If counter is zero, skip
CopyLoop:
lb      $4, 0($2)           // Load 1 byte
sb      $4, 0($3)           // Store 1 byte
addiu   $2, $2, 1           // Increase source pointer by 1 byte
addiu   $3, $3, 1           // Increase destination pointer by 1 byte
addiu   $1, $1, -1          // Decrement counter
bne     $1, $0, CopyLoop    // If not zero, repeat
AfterLoop:
Following instructions
```

- (b) What is the size of the MIPS assembly code you wrote in (a), in bytes? How does it compare to REP MOVSB in x86 (note: REP MOVSB occupies 2 bytes)?

The size of the MIPS assembly code is 4 bytes \times 7 = 28 bytes, as compared to 2 bytes for x86 REP MOVSB.

- (c) Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

```
EAX: 0xccccaaaa
EBP: 0x00002222
ECX: 0xFEE1DEAD
EDX: 0xfed4444
ESI: 0xdecaffff
EDI: 0xdeaddeed
EBP: 0xe0000000
ESP: 0xe0000000
```

Now, consider the MIPS assembly code you wrote in (a). How many total instructions will be executed by your code to accomplish the same function as the single REP MOVSB in x86 accomplishes for the given register state?

The count (value in ECX) is 0xfed1dead = 4276215469. Therefore, the loop body is executed 4276215469 times. As there are 6 instructions in the loop body, total instructions executed = 6 * 4276215469 + 1 = 25657292814 + 1 (beq instruction outside of the loop) = 25657292815.

- (d) Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

EAX: 0xccccaaaa
EBP: 0x00002222
ECX: 0x00000000
EDX: 0xfedd4444
ESI: 0xdecfffff
EDI: 0xdeaddeed
EBP: 0xe0000000
ESP: 0xe0000000

Now, answer the same question in (c) for the above register values.

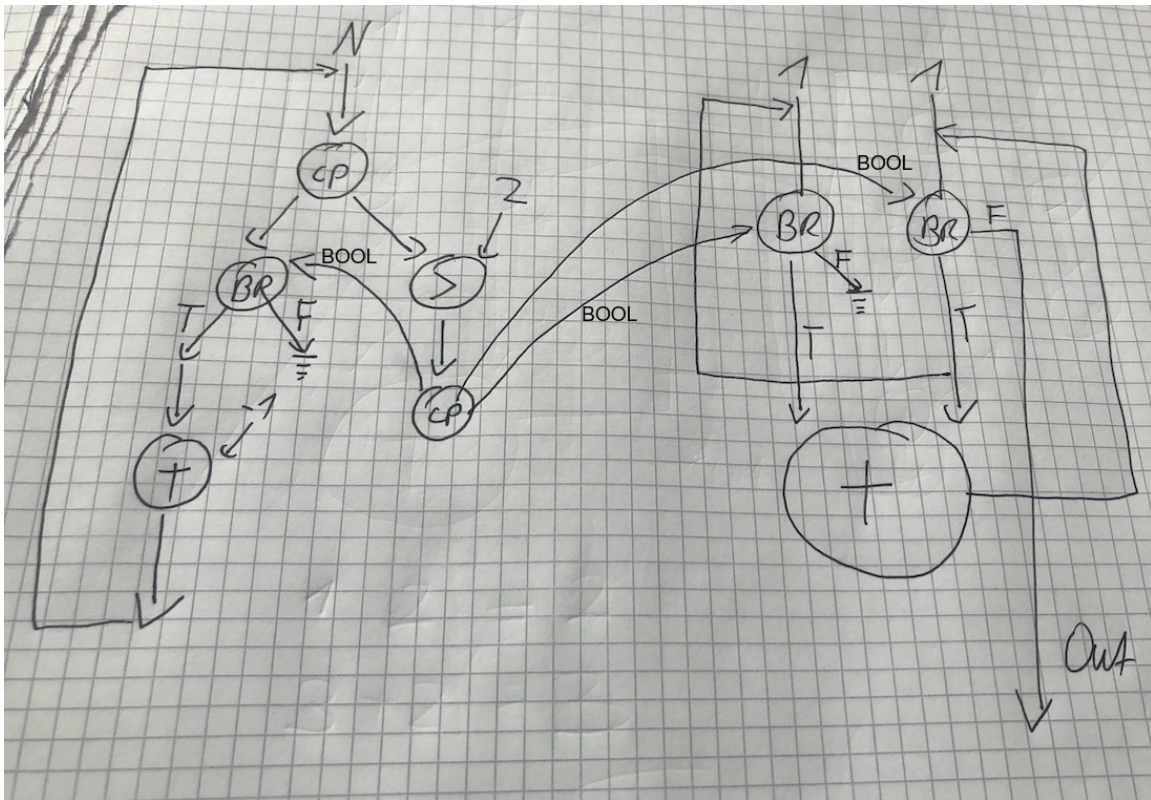
The count (value in ECX) is $0x00000000 = 0$. Therefore, the loop body is executed 0 times. Total instructions executed = 1 (beq instruction outside of the loop).

3 Dataflow (I)

Draw the data flow graph for the `fib(n)` function from Question 2.1. You may use the following data flow nodes in your graph:

- `+` (addition)
- `>` (left operand is greater than right operand)
- `Copy` (copy the value on the input to both outputs)
- `BR` (branch, with the semantics discussed in class, label the True and False outputs)

You can use constant inputs (e.g., 1) that feed into the nodes. Clearly label all the nodes, program inputs, and program outputs. Try to use the fewest number of data flow nodes possible.



4 Dataflow (II)

- We define the *switch node* in Figure 1 to have 2 inputs (**I**, **Ctrl**) and 1 output (**O**). The *Ctrl* input always enters perpendicularly to the switch node. If the *Ctrl* input has a *True* token (i.e., a token with a value of 1), the **O** wire propagates the value on the **I** wire. Else, the 2 input tokens (**I**, **Ctrl**) are consumed, and no token is generated at the output (**O**).
- We define the *inverter node* in Figure 2 to have 1 input (**I**) and 1 output (**O**). The node negates the input token (i.e., $O = !I$).
- We define the *TF node* in Figure 3 to have 3 inputs (I_F , I_T , **Ctrl**) and 1 output (**O**). When **Ctrl** is set to True, **O** takes I_T . When **Ctrl** is set to False, **O** takes I_F .
- The \geq node outputs True only when the left input is greater than or equal to the right input.
- The $+1$ node outputs the input plus one.
- The $+$ node outputs the sum of the two inputs.
- A node generates an output token when tokens exist at *every* input, and *all* input tokens are consumed.
- Where a single wire splits into multiple wires, the token travelling on the wire is replicated to all wires.

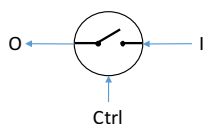


Figure 1: Switch Node

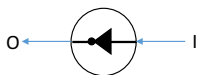


Figure 2: Inverter Node

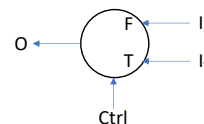


Figure 3: TF Node

Consider the dataflow graph on the following page. Numbers in dashed boxes represent tokens (with the value indicated by the number) in the initial state. The **X** and **Y** inputs automatically produce tokens as soon as the previous token on the wire is consumed. The order of these tokens follows the pattern (*note, the following are all single digit values spaced appropriately for the reader to easily notice the pattern*):

X: 0 01 011 0111 01111

Y: 1 22 333 4444 55555

Consider the dataflow graph on the following page. Please clearly describe the sequence of tokens generated at the output (OUT).

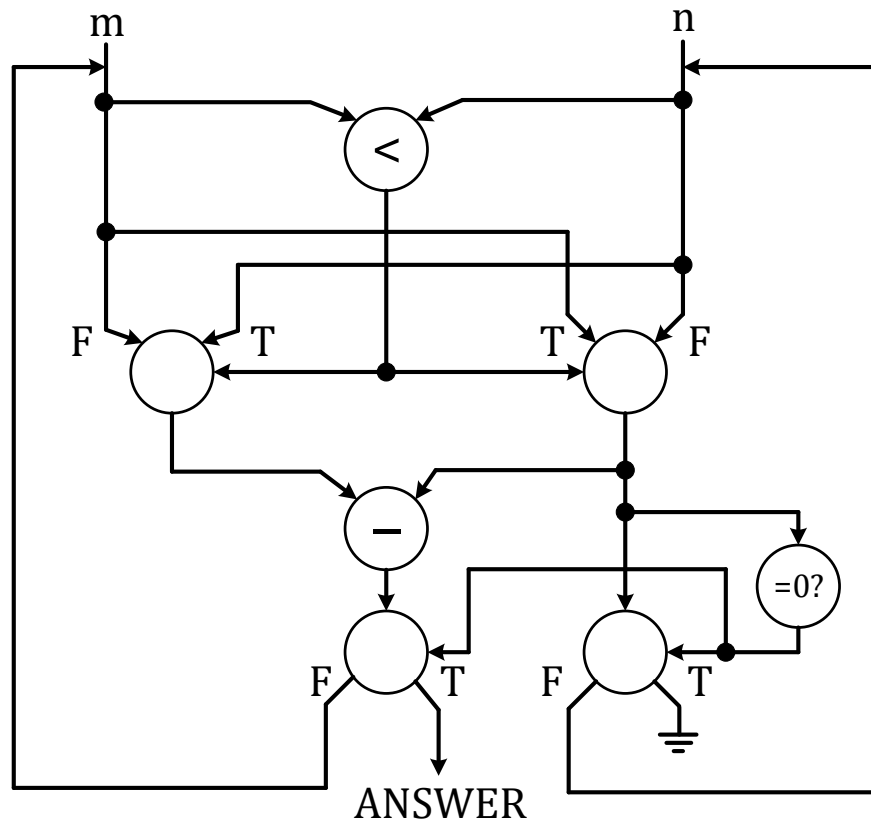
1, 4, 9, 16, 25



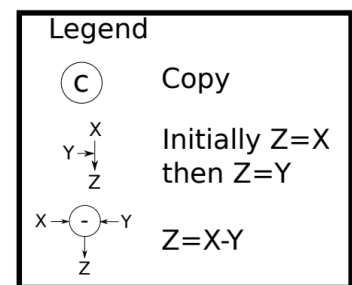
5 Dataflow (III)

- (a) What does the following dataflow program do? Specify clearly in less than 10 words (one could specify this function in three words).

Greatest Common Divisor (this is Euclid's Algorithm)



- Calculates the Hamming distance of A and B.



6 Microarchitecture vs. ISA (I)

- a) Briefly explain the difference between the *microarchitecture* level and the *ISA* level in the transformation hierarchy. What information does the compiler need to know about the microarchitecture of the machine in order to compile a given program correctly?

The ISA level is the interface a machine exposes to the software. The microarchitecture is the actual underlying implementation of the machine. Therefore, the microarchitecture and changes to the microarchitecture are transparent to the compiler/programmer (except in terms of performance), while changes to the ISA affect the compiler/programmer. The compiler does not need to know about the microarchitecture of the machine in order to compile the program correctly

- b) Classify the following attributes of a machine as either a property of its microarchitecture or ISA:

Microarchitecture?	ISA?	Attribute
	✓	The machine does not have a subtract instruction
✓		The ALU of the machine does not have a subtract unit
	✓	The machine does not have condition codes
	✓	A 5-bit immediate can be specified in an ADD instruction
✓		It takes n cycles to execute an ADD instruction
	✓	There are 8 general purpose registers
✓		A 2-to-1 mux feeds one of the inputs to ALU
✓		The register file has one input port and two output ports

7 Microarchitecture vs. ISA (II)

A new CPU has two comprehensive user manuals available for purchase as shown in Table 1.

Manual Title	Cost	Description
the_isa.pdf	CHF 1 million	describes the ISA in detail
the_microarchitecture.pdf	CHF 10 million	describes the microarchitecture in detail

Table 1: Manual Costs

Unfortunately, the manuals are extremely expensive, and you can only afford one of the two. If both manuals might be useful, you would prefer the cheaper one.

For each of the following questions that you would like to answer, decide which manual is more likely to help. *Note: we will subtract 1 point for each **incorrect** answer.*

1. The latency of a branch predictor misprediction.

1. the_isa.pdf 2. the_microarchitecture.pdf

2. The size of a physical memory page.

1. the_isa.pdf 2. the_microarchitecture.pdf

3. The memory-mapped locations of exception vectors.

1. the_isa.pdf 2. the_microarchitecture.pdf

4. The function of each bit in a programmable branch-predictor configuration register.

1. the_isa.pdf 2. the_microarchitecture.pdf

5. The bit-width of the interface between the CPU and the L1 cache.

1. the_isa.pdf 2. the_microarchitecture.pdf

6. The number of pipeline stages in the CPU.

1. the_isa.pdf 2. the_microarchitecture.pdf

7. The order in which loads and stores are executed by a multi-core CPU.

1. the_isa.pdf 2. the_microarchitecture.pdf

8. The memory addressing modes available for arithmetic operations.

1. the_isa.pdf 2. the_microarchitecture.pdf

9. The program counter width.

1. the_isa.pdf 2. the_microarchitecture.pdf

10. The number of cache sets at each level of the cache hierarchy.

1. the_isa.pdf 2. the_microarchitecture.pdf

8 Performance Metrics

- If a given program runs on a processor with a higher frequency, does it imply that the processor always executes more instructions per second (compared to a processor with a lower frequency)? (Use less than 10 words.)

No, the lower frequency processor might have much higher IPC (instructions per cycle).

More detail: A processor with a lower frequency might be able to execute multiple instructions per cycle while a processor with a higher frequency might only execute one instruction per cycle.

- If a processor executes more of a given program's instructions per second, does it imply that the processor always finishes the program faster (compared to a processor that executes fewer instructions per second)? (Use less than 10 words.)

No, because the former processor may execute many more instructions.

More detail: The total number of instructions required to execute the full program could be different on different processors.

9 Performance Evaluation (I)

Your job is to evaluate the potential performance of two processors, each implementing a different ISA. The evaluation is based on its performance on a particular benchmark. On the processor implementing ISA *A*, the best compiled code for this benchmark performs at the rate of 10 IPC. That processor has a 500 MHz clock. On the processor implementing ISA *B*, the best compiled code for this benchmark performs at the rate of 2 IPC. That processor has a 600 MHz clock.

- What is the performance in Millions of Instructions per Second (MIPS) of the processor implementing ISA *A*?

$$\text{ISA } A: 10 \frac{\text{instructions}}{\text{cycle}} * 500,000,000 \frac{\text{cycle}}{\text{second}} = 5000 \text{ MIPS}$$

- What is the performance in MIPS of the processor implementing ISA *B*?

$$\text{ISA } B: 2 \frac{\text{instructions}}{\text{cycle}} * 600,000,000 \frac{\text{cycle}}{\text{second}} = 1200 \text{ MIPS}$$

- Which is the higher performance processor: *A* *B* Don't know
Briefly explain your answer.

Don't know.

The best compiled code for each processor may have a different number of instructions.

10 Performance Evaluation (II)

You are the leading engineer of a new processor. Both the design of the processor and the compiler for it are already done. Now, you need to decide if you will send the processor to manufacturing at its current stage or if you will delay the production to introduce last-minute improvements to the design. To make the decision, you meet with your team to brainstorm about how to improve the design. Together, after profiling the target applications for the processor, you come up with two options:

- **Keep the current project.** For version A of the processor, the clock frequency is 600 MHz, and the following measurements are obtained:

Instruction Class	CPI	Frequency of Occurrence
A	2	40%
B	3	25%
C	3	25%
D	7	10%

- **Include optimizations to the design.** For version B of the processor, the clock frequency is 700 MHz. The ISA for processor B includes three new types of instructions. Those three new types of instructions increase the total number of executed instructions for processor B by 50%, in comparison to processor A. The following measurements are obtained:

Instruction Class	CPI	Frequency of Occurrence
A	2	15%
B	2	15%
C	4	10%
D	6	10%
E	1	10%
F	2	20%
G	2	20%

- (a) What is the CPI of each version? Show your work.

$$CPI_A: \quad CPI_A = 2 \times 0.4 + 3 \times 0.25 + 3 \times 0.25 + 7 \times 0.1$$

$$CPI_A = 3.$$

$$CPI_B: \quad CPI_B = 2 \times 0.15 + 2 \times 0.15 + 4 \times 0.1 + 6 \times 0.1 + 1 \times 0.1 + 2 \times 0.2 + 2 \times 0.2$$

$$CPI_B = 2.5.$$

- (b) What are the MIPS (Million Instructions Per Second) of each version? Show your work.

$$MIPS_A: \quad MIPS_A = \frac{600MHz}{3*10^6}$$

$$MIPS_A = 200.$$

$$MIPS_B: \quad MIPS_B = \frac{700MHz}{2.5*10^6}$$

$$MIPS_B = 280.$$

- (c) Considering your team is aiming to release to the market the processor that gives better performance when executing the target application, which processor version will you choose as the final design? Show your work.

Processor A.

Explanation:

We calculate the execution time for each processor, $Time = N_{instr.} \times CPI \times \frac{1}{clockfrequency}$ Since the compiler for processor B generates 50% more instructions than the compiler for processor A, the total execution time for processor B is larger than the total execution time for processor A.

$$Time_A = N_{instr.} \times 3 \times \frac{1}{600*10^6}$$

$$Time_B = 1.5N_{instr.} \times 2.5 \times \frac{1}{700*10^6}$$

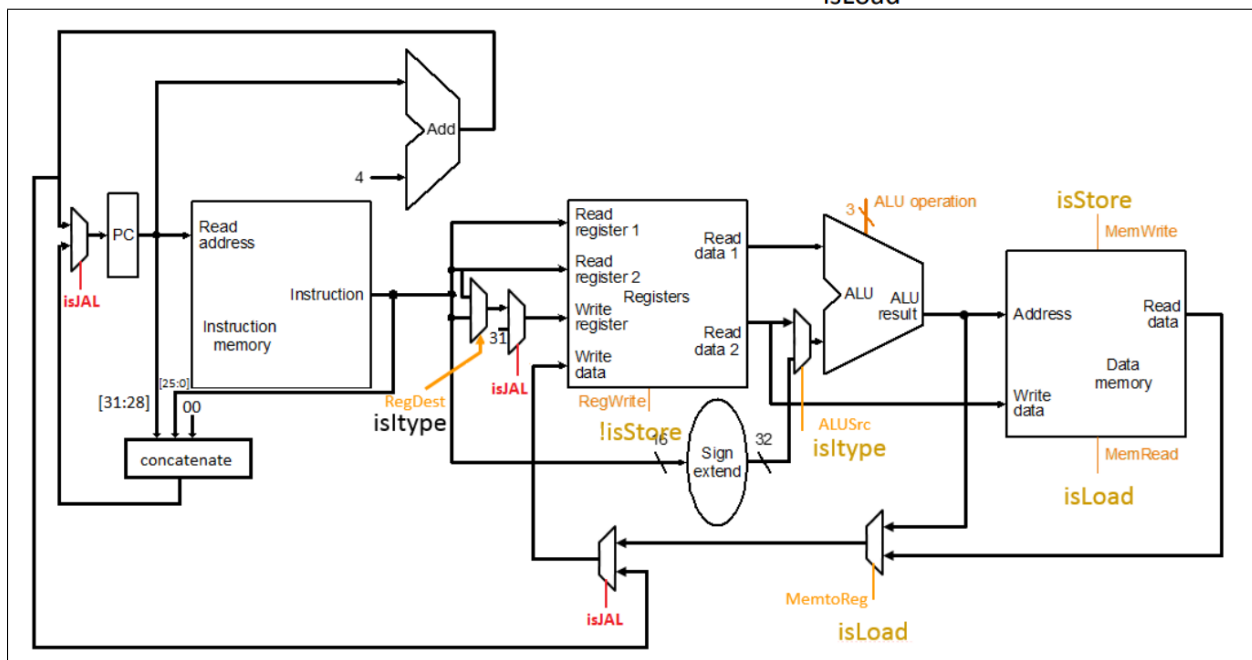
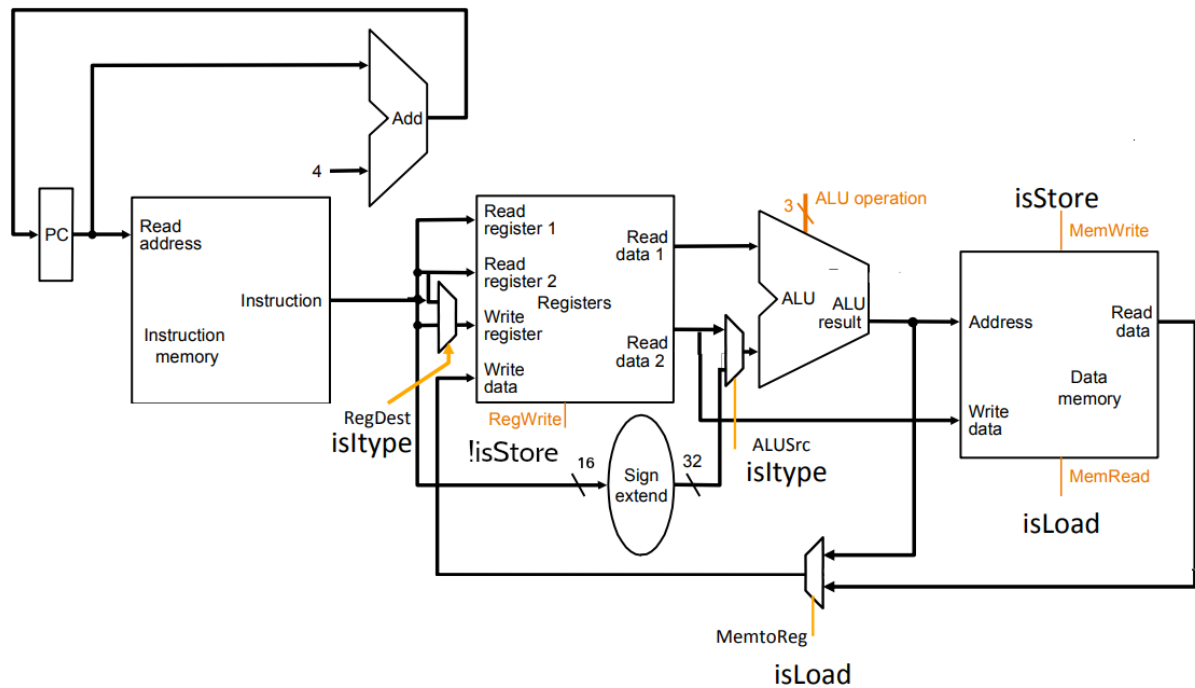
11 Single-Cycle Processor Datapath

In this problem, you will modify the single-cycle datapath we built up in Lecture 11 to support the JAL instruction. The datapath that we will start with is provided below. Your job is to implement the necessary data and control signals to support the JAL instruction, which we define to have the following semantics:

$$\text{JAL : } R31 \leftarrow \text{PC} + 4$$

$$\text{PC} \leftarrow \text{PC}_{31:28} \parallel \text{Immediate} \parallel 0^2$$

Add to the datapath on the next page the necessary data and control signals to implement the JAL instruction. Draw and label all components and wires very clearly (give control signals meaningful names; if selecting a subset of bits from many, specify exactly which bits are selected; and so on).



12 REP MOVSB

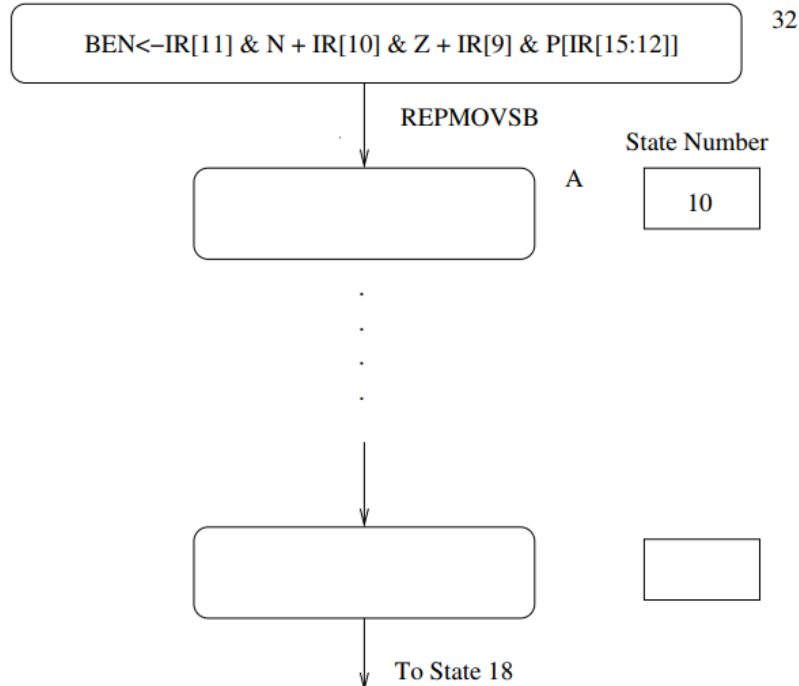
Let's say you are the lead architect of the next flagship processor at Advanced Number Devices (AND). You have decided that you want to use the LC-3b ISA for your next product, but your customers want a smaller semantic gap and marketing is on your case about it. So, you have decided to implement your favorite x86 instruction, REP MOVSB, in LC-3b.

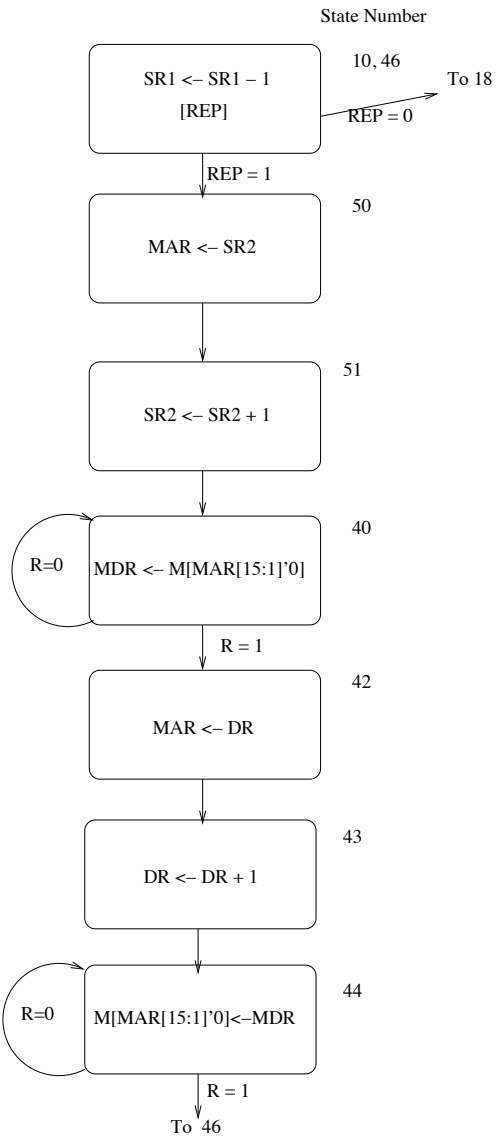
Specifically, you want to implement the following definition for REP MOVSB (in LC-3b parlance): REP-MOVSB SR1, SR2, DR which is encoded in LC-3b machine code as:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1010				DR			SR1			0	0	0	SR2		

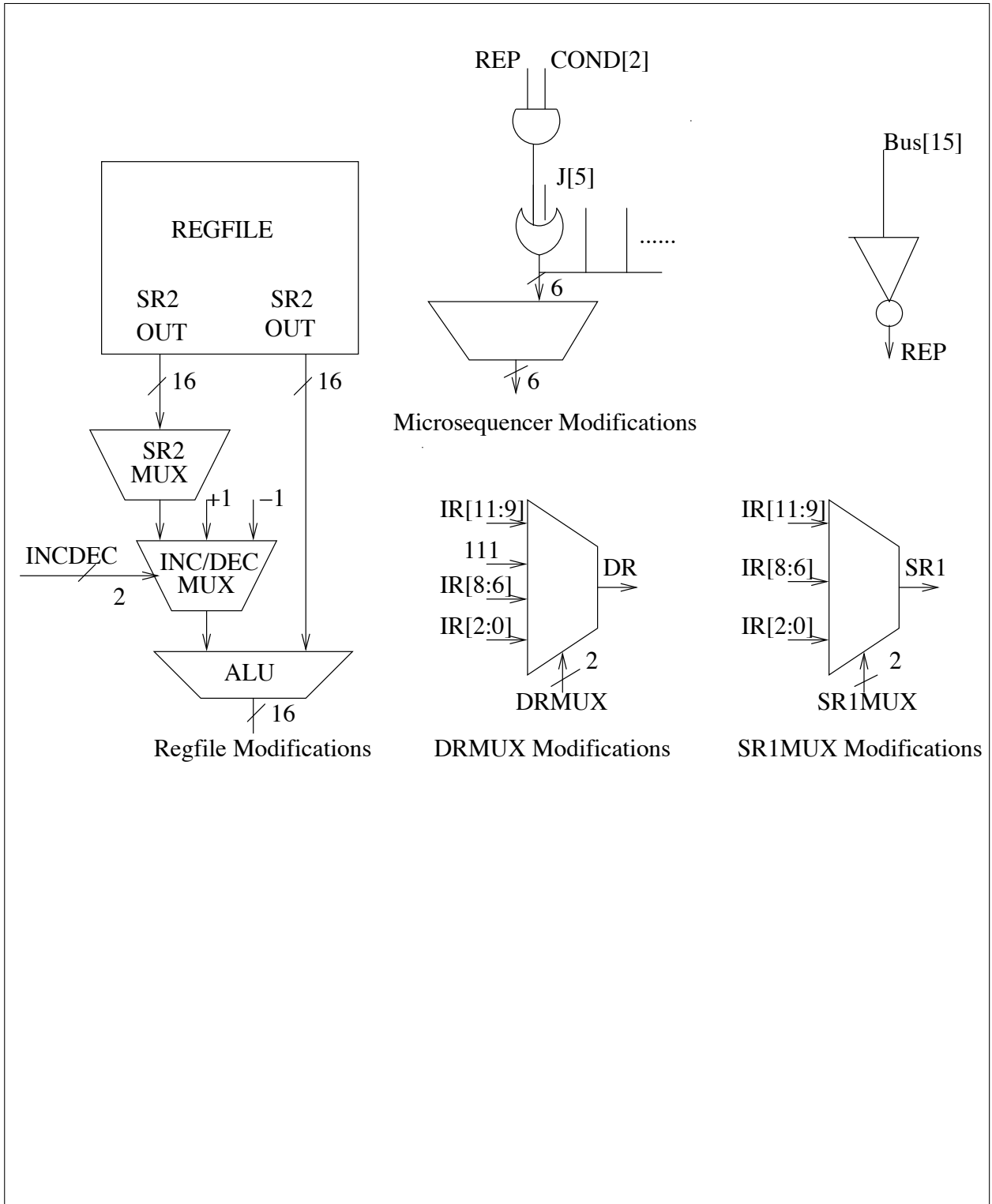
REPMOVSB uses three registers: SR1 (count), SR2 (source), and DR (destination). It moves a byte from memory at address SR2 to memory at address DR, and then increments both pointers by one. This is repeated SR1 times. Thus, the instruction copies SR1 bytes from address SR2 to address DR. Assume that the value in SR1 is greater than or equal to zero.

- (a) Complete the state diagram shown below, using the notation of the LC-3b state diagram. Describe inside each bubble what happens in each state and assign each state an appropriate state number. Add additional states not present in the original LC-3b design as you see fit.





- (b) Add to the LC-3b datapath any additional structures and any additional control signals needed to implement REPMOVS. Clearly label your additional control signals with descriptive names. Describe what value each control signal would take to control the datapath in a particular way.



- (c) Describe any changes you need to make to the LC-3b microsequencer. Add any additional logic and control signals you need. Clearly describe the purpose and function of each signal and the values it would take to control the microsequencer in a particular way.

Additional control signals

- INCDEC/2: PASSSR2, +1, -1
- DRMUX/2:
 - IR[11:9] ;destination IR[11:9]
 - R7 ;destination R7
 - IR[8:6] ;destination IR[8:6]
 - IR[2:0] ;destination IR[2:0]
- SR1MUX/2:
 - IR[11:9] ;source IR[11:9]
 - IR[8:6] ;source IR[8:6]
 - IR[2:0] ;source IR[2:0]
- COND/3:
 - COND0: Unconditional
 - COND1: Memory Ready
 - COND2: Branch
 - COND3: Addressing Mode
 - COND4: Repeat

MIPS Instruction Summary

Opcode	Example Assembly	Semantics
add	add \$1, \$2, \$3	$\$1 = \$2 + \$3$
sub	sub \$1, \$2, \$3	$\$1 = \$2 - \$3$
add immediate	addi \$1, \$2, 100	$\$1 = \$2 + 100$
add unsigned	addu \$1, \$2, \$3	$\$1 = \$2 + \$3$
subtract unsigned	subu \$1, \$2, \$3	$\$1 = \$2 - \$3$
add immediate unsigned	addiu \$1, \$2, 100	$\$1 = \$2 + 100$
multiply	mult \$2, \$3	hi, lo = $\$2 * \3
multiply unsigned	multu \$2, \$3	hi, lo = $\$2 * \3
divide	div \$2, \$3	lo = $\$2 / \3 , hi = $\$2 \bmod \3
divide unsigned	divu \$2, \$3	lo = $\$2 / \3 , hi = $\$2 \bmod \3
move from hi	mfhi \$1	$\$1 = \text{hi}$
move from low	mflo \$1	$\$1 = \text{lo}$
and	and \$1, \$2, \$3	$\$1 = \$2 \& \$3$
or	or \$1, \$2, \$3	$\$1 = \$2 \$3$
and immediate	andi \$1, \$2, 100	$\$1 = \$2 \& 100$
or immediate	ori \$1, \$2, 100	$\$1 = \$2 100$
shift left logical	sll \$1, \$2, 10	$\$1 = \$2 \ll 10$
shift right logical	srl \$1, \$2, 10	$\$1 = \$2 \gg 10$
load word	lw \$1, 100(\$2)	$\$1 = \text{memory}[\$2 + 100]$
store word	sw \$1, 100(\$2)	$\text{memory}[\$2 + 100] = \1
load upper immediate	lui \$1, 100	$\$1 = 100 \ll 16$
branch on equal	beq \$1, \$2, label	if ($\$1 == \2) goto label
branch on not equal	bne \$1, \$2, label	if ($\$1 \neq \2) goto label
set on less than	slt \$1, \$2, \$3	if ($\$2 < \3) $\$1 = 1$ else $\$1 = 0$
set on less than immediate	slti \$1, \$2, 100	if ($\$2 < 100$) $\$1 = 1$ else $\$1 = 0$
set on less than unsigned	sltu \$1, \$2, \$3	if ($\$2 < \3) $\$1 = 1$ else $\$1 = 0$
set on less than immediate unsigned	sltui \$1, \$2, 100	if ($\$2 < 100$) $\$1 = 1$ else $\$1 = 0$
jump	j label	goto label
jump register	jr \$31	goto \$31
jump and link	jal label	$\$31 = \text{PC} + 4$; goto label